# The Scratch Programming Language and Environment

JOHN MALONEY, MITCHEL RESNICK, NATALIE RUSK,
BRIAN SILVERMAN, and EVELYN EASTMOND
Massachusetts Institute of Technology

Scratch is a visual programming environment that allows users (primarily ages 8 to 16) to learn computer programming while working on personally meaningful projects such as animated stories and games. A key design goal of Scratch is to support self-directed learning through tinkering and collaboration with peers. This article explores how the Scratch programming language and environment support this goal.

Categories and Subject Descriptors: K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Scratch, visual programming language, programming language, programming environment

**ACM Reference Format:**
Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. 2010. The scratch programming language and environment. ACM Trans. Comput. Educ. 10, 4, Article 16 (November 2010), 15 pages. DOI = 10.1145/1868358.1868363. http://doi.acm.org/10.1145/1868358.1868363.

## 1. INTRODUCTION

Scratch is a visual programming environment that lets users create interactive, media-rich projects. People have created a wide range of projects with Scratch, including animated stories, games, online news shows, book reports, greeting cards, music videos, science projects, tutorials, simulations, and sensor-driven art and music projects (Figure 1).

The Scratch application is used to create projects containing media and scripts. Images and sounds can be imported or created in Scratch using a built-in paint tool and sound recorder. Programming is done by snapping together

Author's address: J. Maloney, MIT Media Laboratory, E14-464B, 75 Amherst St., Cambridge, MA 02139; email: jmaloney@media.mit.edu.

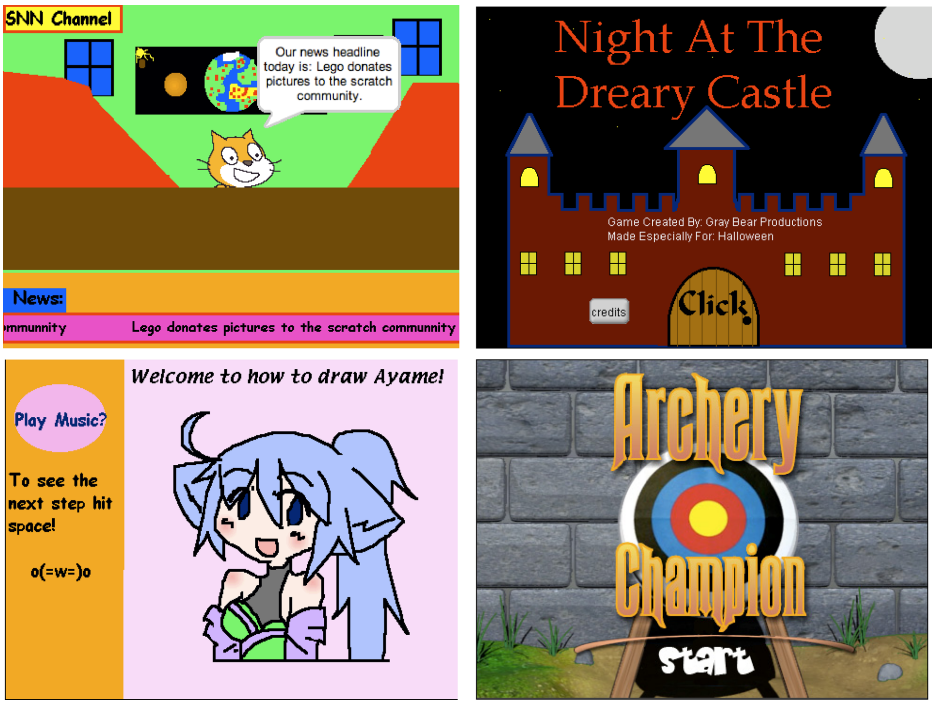


Fig. 1. Screenshots from Scratch projects created by users, including a news show, an interactive story, a drawing tutorial, and a game.

colorful command blocks to control 2-D graphical objects called sprites moving on a background called the stage. Scratch projects can be saved to the file system or shared on the Scratch Web site.

The original design of Scratch was motivated by the needs and interests of young people (ages 8 to 16) at after-school computer centers such as the Intel Computer Clubhouses [Resnick et al. 2003]. Scratch added programmability to media-manipulation activities that are popular in youth culture, and it encouraged young people to learn through exploration and peer sharing, with less focus on direct instruction than other programming languages. Initially, Scratch was used primarily in informal learning settings such as community centers, after-school clubs, libraries, and homes, but increasingly it is used in schools as well.

The Scratch project began in 2003, and the Scratch software and Web site[1] were publicly launched in 2007. Scratch is free, available in nearly 50 languages, and more than two million copies have been downloaded from the Scratch Web site. In addition, Scratch software is often redistributed by school systems and educational organizations. For example, Scratch is distributed by One Laptop Per Child and has been shipped on hundreds of thousands of XO laptop computers. Since the launch, more than a million Scratch projects have

[1]See scratch.mit.edu.

been uploaded to the Web site by more than 120,000 users. Roughly 1500 new projects are uploaded to the Web site every day—on average, more than one new project every minute.

Scratch builds on the constructionist ideas of Logo [Kafai and Resnick 1996; Papert 1980] and Etoys [Kay 2010; Steinmetz 2002]. To help users make their projects personally engaging, motivating, and meaningful, Scratch makes it easy to import or create many kinds of media (images, sounds, music). The Scratch Web site provides a social context for Scratch users, allowing users to share their Scratch projects, receive feedback and encouragement from their peers, and learn from the projects of others [Resnick et al. 2009].

A key goal of Scratch is to introduce programming to those with no previous programming experience. This goal drove many aspects of the Scratch design. Some of the design decisions are obvious, such as the choice of a visual blocks language, the single-window user interface layout, and the minimal command set. Others are less obvious, such as how the target audience influenced the type system and the approach to error handling. This article explores aspects of the Scratch programming environment and language design that make it easier for young people to explore, express themselves, and learn.

## 2. PROGRAMMING ENVIRONMENT

Many users learn Scratch as they go, trying commands from the palette or exploring code from existing projects. To encourage such self-directed learning, the Scratch programming environment was designed to invite scripting, provide immediate feedback for script execution, and make execution and data visible.

### 2.1 Single-Window User Interface

The Scratch user interface strives to make navigation easy. It uses a single-window, multi-pane design to ensure that key components are always visible. Scratch avoids floating palettes, which can get buried, and minimizes the use of panes that show only on demand.

Figure 2 shows the Scratch window, which has four main panes. The left pane is the command palette with buttons to select categories. The middle pane shows the scripts for the currently selected sprite, with folder tabs to view and edit the costumes (images) and sounds owned by that sprite. The large pane on the upper right is the stage, where the action happens. The bottom-right pane shows thumbnails of all sprites in the project, with the currently selected sprite highlighted.

To invite scripting, the command palette is always visible. The commands are divided into eight categories such as Motion, Looks, Sound, and Control. This avoids long, potentially overwhelming, lists of commands: in most palettes, all the commands can be viewed without scrolling. In each category, the most self-explanatory and useful commands appear near the top of the command palette. Command blocks are color-coded by category, helping users find related blocks.

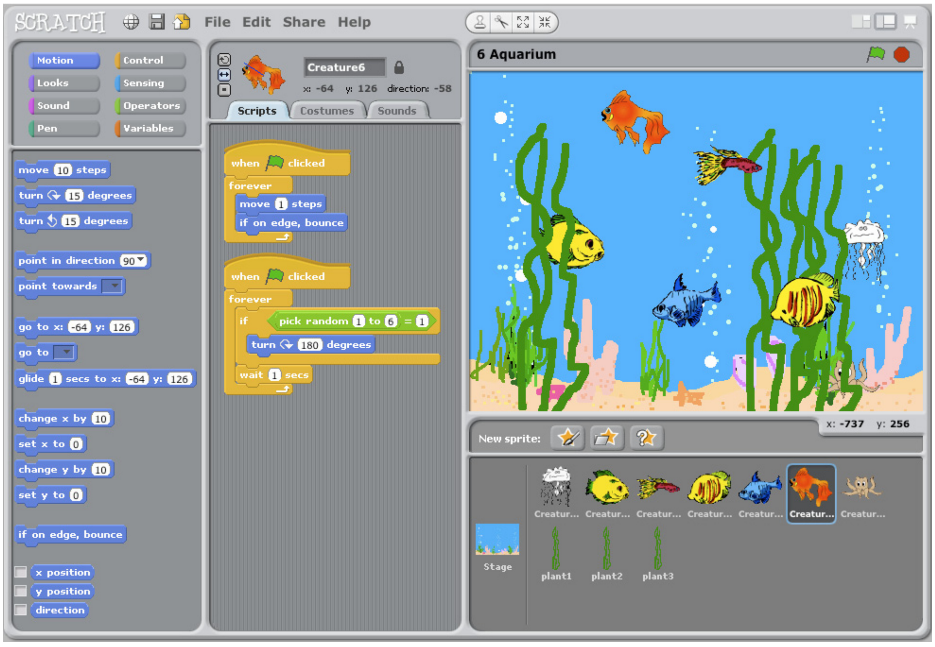


Fig. 2. The Scratch user interface.

## 2.2 Liveness and Tinkerability

A key feature of Scratch is that it is always *live* [Maloney and Smith 1995]. There is no compilation step or edit/run mode distinction. Users can click on a command or program fragment at any time to see what it does. In fact, they can even change parameters or add blocks to a script while it is running. By eliminating potentially jarring mode switches and compilation pauses, Scratch helps users stay engaged in testing, debugging, and improving their projects.

We say that Scratch is *tinkerable* because it lets users experiment with commands and code snippets the way one might tinker with mechanical or electronic components. Tinkerability encourages hands-on learning and supports a bottom-up approach to writing scripts where small chunks of code are assembled and tested, then combined into larger units.

Tinkerability helps users discover the functionality of blocks. A block can be tested by clicking on it, even in the palette. Function blocks show their return value in a cartoon-like "talk bubble" (Figure 3). To help users more easily explore what blocks do, each block comes with default parameters that give an illuminating demonstration of what that block does. Scratch has help screens for every command, accessible via the right-button menu, but many users learn about commands just by trying them.

Scratch does not require that the user create complete scripts before running the projects. Program fragments can be left in the scripting pane and are saved with the project. Such fragments play a role similar to commented-out code in


Fig. 3. Blocks can be tested simply by clicking on them. A white border indicates that a block or stack is running. Function blocks show their output in a talk bubble.

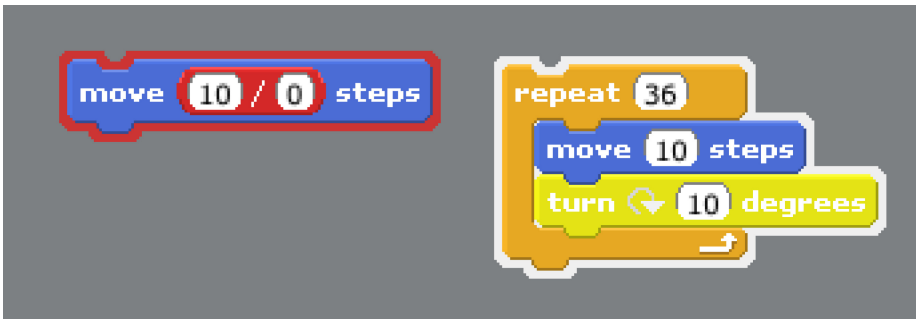


Fig. 4. Feedback for an error is red (left). When single-stepping is enabled, the currently executing block glows a bright yellow (right).

a text-based language. When troubleshooting, a long script can be broken into pieces and each piece tested independently.

### 2.3 Making Execution Visible

Scratch provides visual feedback to show script execution. When a script is running, it is surrounded by a glowing white border (Figure 3). This feedback helps the user understand when scripts are triggered and how long they run. If a script encounters an error (e.g., dividing by zero), the border turns red and the block that caused the error is highlighted in red (Figure 4).

Scratch can also show command sequencing and flow of control. Enabling *single-stepping* (selected from a menu) causes blocks to flash as they run (Figure 4). Even when single-stepping mode is not enabled, Scratch updates the display after every command. Seeing the effect of every command, even if only as a brief flash on the screen, provides important visual clues when troubleshooting.

### 2.4 No Error Messages

When people play with LEGO® bricks, they do not encounter error messages. Parts stick together only in certain ways, and it is easier to get things right than wrong. The brick shapes suggest what is possible, and experimentation and experience teaches what works.


Fig. 5. Scratch variable monitors (left) and list monitor (right). The optional slider allows a variable to be used as a control.

Similarly, Scratch has no error messages. Syntax errors are eliminated because, like LEGO® bricks, blocks fit together only in ways that make sense. But Scratch also strives to eliminate run time errors by making all blocks be *failsoft*. Rather than failing with an error message, every block attempts to do something sensible even when presented with out-of-range inputs. For example, the "set size" block bounds the range of its parameter so that it cannot make the sprite excessively large (possibly exceeding system limits) or invisibly small.

Of course, eliminating error messages does not eliminate errors. The user must still think carefully to write scripts that do what they want and must trouble-shoot scripts that do not work as expected. However, even when a script does not do the right thing, it does something, and that is a good start. A program that runs, even if it is not correct, feels closer to working than a program that does not run (or compile) at all.

### 2.5 Making Data Concrete

In most text-based programming languages, variables are invisible, abstract, and difficult to understand. Like earlier systems such as Boxer [diSessa and Abelson 1986], Scratch turns variables into concrete objects that the user can see and manipulate, making them easier to understand through tinkering and observation.

In Scratch, a variable can appear on the stage as a *variable monitor* (Figure 5). Monitors allow users to see the effect of commands such as "change x by 1", helping them build a mental picture of how variables work. But monitors are not only an aid to understanding; they are also useful for their own sake as readouts (e.g., to display the score in a game) or, using the optional slider, as controls.

Scratch also has monitors for lists. When a list monitor is on the stage, quick animations show the effects of list operations. For example, when a list element is accessed, the index of that element flashes.

Small design details can make a big difference. In early versions of Scratch, a newly created variable was not displayed on the stage, and the gesture to make that happen was not obvious. Many users did not discover variables, even

when they needed them. After changing the design so that a newly-created variable is immediately displayed, many more users started using variables.

### 2.6 Minimizing the Command Set

Scratch strives to minimize the number of command blocks while still supporting a wide range of project types. One might argue that flexibility, programmer convenience, and extra features are more important than a small command set. However, in Scratch, unlike a text-based language, every command consumes screen space in the command palettes, so there is a higher "cost" to increasing the command set. Adding more commands requires either adding more categories or forces the user to scroll down to see all the commands within a given category. Either way, a larger command set makes it harder to find a given command in the palettes.

Scratch 1.0 had 92 command blocks. As Scratch has evolved, it has been a constant struggle to keep down the number of commands. Every release adds new features and new commands. Occasionally, a command is withdrawn. (Withdrawn commands, called *obsolete blocks*, are still supported by the runtime system to allow old projects to run.) However, more commands have been added than removed. Scratch 1.4 has 125 command blocks, although some of them do not appear until needed.

One strategy for reducing the number of command blocks is to group a set of related operations into a single block with a drop-down menu to select the specific operation. Examples of this technique include the scientific math function block (a dozen math functions) and the image effect blocks (seven image effects), both of which were originally collections of individual blocks.

Another strategy is to make sets of command blocks appear on demand when they are first needed. For example, the five blocks to control the motor of the LEGO WeDo® robotics kit motor appear when a WeDo USB hub is plugged into the computer. Similarly, the blocks to access variables and lists appear only after a variable or list has been created.

## 3. PROGRAMMING LANGUAGE

This section examines the design of the Scratch programming language: its syntax (i.e., visual blocks), type system, object model, inter-object communications, and approach to concurrency.

### 3.1 Syntax

Scratch scripts are built by snapping together blocks representing statements, expressions, and control structures. The shapes of the blocks suggest how they fit together, and the drag-and-drop system refuses to connect blocks in ways that would be meaningless. In Scratch, the visual grammar of block shapes and their combination rules play the role of syntax in a text-based language.

There are four kinds of Scratch blocks: command blocks, function blocks, trigger blocks, and control structure blocks, as shown in Table I. When command blocks are snapped together to create a sequence of commands, or *stack*, the notches and bumps fit together like puzzle pieces.

Table I. Scratch Block Types

| | |
|---|---|
|   | A *command* block has a notch on the top and a matching bump on the bottom. Command blocks can be joined to create a sequence of commands called a *stack*. |
|   | A *function* block returns a value. Function blocks do not have notches. |
|   | A *trigger block* has a rounded top. It runs the tack below it when the triggering event occurs. |
|   | *Control structure* command blocks have openings to hold nested command sequences. |

Control structure blocks are a kind of command block with one or more nested command sequences. The form of control structure blocks makes them easy to use. In most text-based languages, the closing delimiters for control structures can be omitted or misplaced, leading to errors. In Scratch, a control structure block is an indivisible unit. The closing arm of a loop or conditional block is part of the block itself—it cannot be misplaced—and the nesting of the enclosed command sequence is manifest. Instructors using Scratch as a quick introduction to programming before switching to a text-based language report that some students continue to "think in Scratch blocks" as a form of pseudocode, even after moving to the text-based language [Malan and Leitner 2007].

Command blocks are like the statements of a text-based language; function blocks are like operators. Function blocks are not joined in linear sequences like command blocks. Instead, they are used as arguments to commands and nested together to build expressions.

Trigger blocks connect events (such as startup, mouse clicks, and key presses) to the stacks that handle those events. For example, all stacks starting with a green flag trigger block are run when the user clicks the start button.

Some blocks have embedded *parameter slots*. The shape of a parameter slot shows the parameter type: number, string, boolean, etc. Some parameter slots (ones with a white background) allow the user to enter a value from the keyboard. Others have drop-down menus or color choosers. Most parameter slots can accept a function block.

When assembling scripts, Scratch only allows blocks to be connected in meaningful ways. A command block connects when dropped into command sequence, but a function block will not connect if dropped in the same place. As the user drags a block, Scratch gives visual feedback showing possible sequence insertion points (command blocks) or parameter slot targets (function blocks).

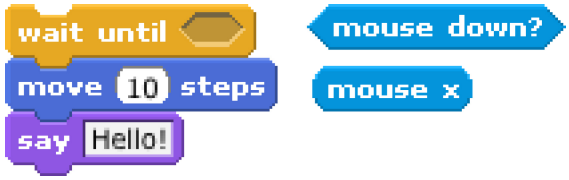


Fig. 6. Shapes indicate type. On the left, command blocks with parameter slots for boolean, number, and string parameters. On the right, boolean and number function blocks.


Fig. 7. Visual feedback while dragging a function block. On the left, the white highlight shows that a boolean block can be inserted into the boolean slot. On the right, absence of the highlight shows that a number block cannot be inserted.

Disassembling stacks is easy. Grabbing the top block of a stack drags the entire stack; grabbing a block in the middle of a stack detaches that block and any blocks below it. Using the blocks editor feels natural and easy, and users often discover how to use it without being told.

### 3.2 Data Types

Scratch has three first-class data types: boolean, number, and string. These are the only data types that can be used in expressions, stored in variables, or returned by built-in functions. In the Scratch visual language, the shape of a parameter slot indicates the data type expected and the shape of a function block indicates the type returned (Figure 6). While there are three parameter slot shapes, there are only two function block shapes: boolean and number/string. This is a consequence of the fact that Scratch variables are untyped and can contain either numbers or a strings.

The Scratch editor only allows a function block to be inserted into a parameter slot if the result would not violate the data typing constraints (Figure 7). Boolean parameter slots are the most strict, accepting only boolean function blocks. Number and string parameter slots are less strict. They accept a function block of any type, coercing the parameter to the target type if necessary.

A Scratch variable can hold values of any data type. This avoids requiring that the user specify the type of a variable when it is created. Scratch automatically converts between numbers and strings depending on context. For example, if the string “123” is passed to an arithmetic operation, it is converted

to a number, while if a number is passed to the "say" command, it is converted to a string. Given the goals of Scratch, automatic conversion is preferable to requiring the user to explicitly convert between types.

Scratch currently supports only three first-class data types. How might the visual grammar be extended to handle additional first-class types in the future? One approach would be to create a new slot/function shape for each new type. But if many types were added, that approach could lead to visual clutter and potential confusion. An alternative approach is to have all new types share the same rounded shape already used for numbers and strings, consistent with Scratch's untyped variables. This design choice mirrors the choice between static and dynamic typing in text-based languages.

### 3.3 Sprites: The Scratch Object Model

Sprites are objects: they encapsulate state (variables) and behavior (scripts). However, since Scratch has neither classes nor inheritance, it is an object-based language but not an object-oriented one. (A language must support inheritance to be called object-oriented [Wegner 1987].)

Commands operate only on the sprite in which they appear; one sprite cannot invoke a command such as "move" on a different sprite. In object-oriented terms, the implicit receiver of every command is the sprite in which it appears. (An early prototype of Scratch allowed cross-sprite commands, but users found that confusing.)

Every sprite has its own independent set of scripts. This design involves a tradeoff. On one hand, it is easy to understand. The scripts in a given sprite tell the entire story about that sprite's behavior; the user need not look up the class hierarchy or follow a prototype chain to find inherited scripts. Furthermore, code changes are localized: editing a script affects only the sprite in which that script appears, whereas in other languages editing an inherited method can have far-reaching and possibly unexpected consequences.

On the other hand, without classes or some other code-sharing mechanism, it is more work to manage multiple sprites with identical behavior, such as the bricks in a Breakout game. In such cases, the user typically creates a single sprite with the desired behavior, then uses the stamp tool to make as many copies as desired. If the behavior of the sprites needs to be changed after the copies have been made, the user can either make the same change in every copy or delete all but one of the copies, edit that sprite's scripts, and then make a new set of copies.

Manually copying sprites grows tedious. Advanced Scratch users often ask for a command to copy, or *clone*, a sprite under program control. We tried several forms of cloning in earlier versions of Scratch, but discovered three complications: (1) a run-away program can overrun the stage with clones, making the system unresponsive; (2) using the current broadcast mechanism, it was difficult to invoke a script on only the new clone; and (3) it required extra logic to destroy the clone when it was no longer needed. The second two problems tend to exacerbate the first. We are currently exploring a new clone mechanism that addresses these issues.

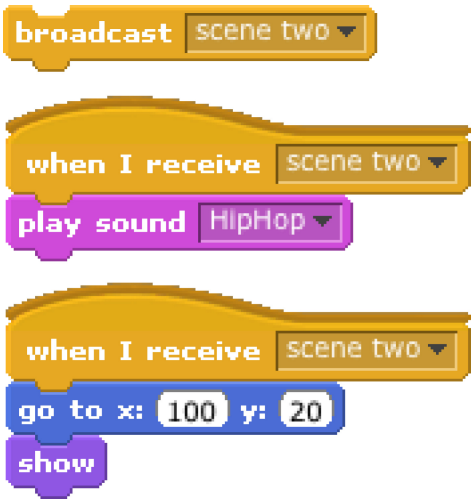


Fig. 8. A broadcast command and two scripts it triggers.

### 3.4 Inter-Sprite Communications and Sharing

As mentioned, sprites cannot call each other's scripts directly. Instead, Scratch uses a *broadcast* mechanism to support inter-sprite communication and synchronization. Any sprite can broadcast a message (an arbitrary string). A broadcast triggers all scripts in all sprites that begin with a matching "when I receive <msg>" trigger block. For example, in response to the broadcast message "scene two," several sprites representing the characters in a story might start acting out that scene (Figure 8).

The Scratch broadcast model is *one-to-many*, because a given broadcast can trigger many scripts (possibly in multiple sprites); *loosely-coupled*, because it does not matter how many receivers there are; and *asynchronous*, because the "broadcast" command does not wait until the triggered scripts complete. A "broadcast" can start scripts that loop forever, similar to starting a thread. Scratch also has a synchronous variant of broadcast that waits until all triggered scripts have completed.

There are two benefits to not allowing sprites to control each other directly. First, when trying to understand why a sprite does something (e.g., moves in a certain way), the scope of possibilities is limited to scripts within that sprite itself. Second, and more importantly, because sprites are self-sufficient and only loosely coupled to other sprites, they can be moved between projects without breaking dependencies. This allows sprites to be shared.

Sharing sprites encourages code reuse and collaboration. For example, a sprite with generic arrow key motion controls could be shared so that other Scratch users can import that sprite into their own projects. In the process of adapting a sprite for their own purposes, users are exposed to new commands and programming techniques. Sharing sprites also fosters collaboration: when users are working together, each of them can independently develop sprites and then combine those sprites to create the final project.

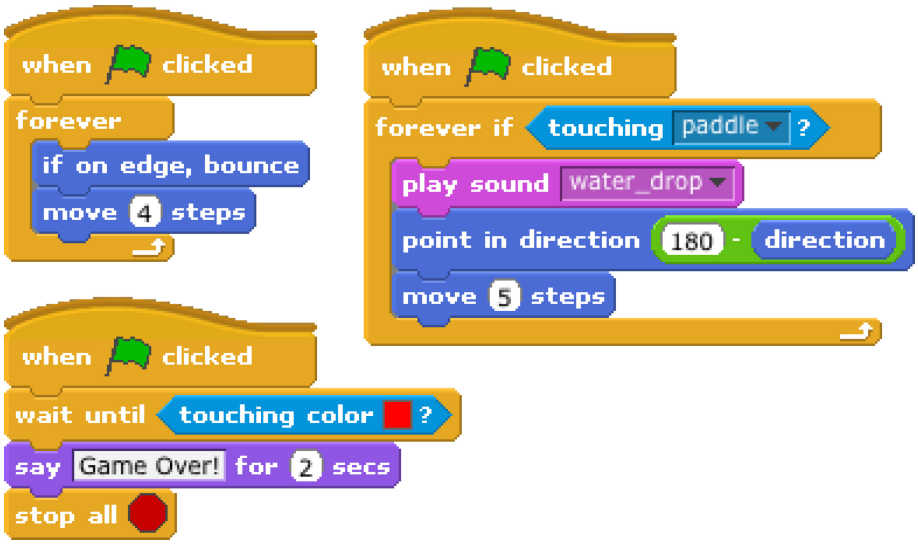


Fig. 9. Three concurrent scripts for the ball in a paddle game.

### 3.5 Procedures

Early versions of Scratch had a mechanism for creating procedures. In early field tests, however, many users were confused by procedures since they seemed very similar to broadcasts—both involved associating a name with a collection of commands. In the interest of simplicity and minimalism, procedures were removed from the language before Scratch was officially released, and Scratch has gotten along surprising well without them.

Yet procedural abstraction is one of the "powerful ideas" of computer science and procedures have practical value as a way to structure code as projects grow. The Scratch team is considering reintroducing procedures as a way for users to define their own command blocks. Other researchers have been developing a variant of Scratch that supports not only procedures but first-class functions, closures, and a complete functional programming system [Harvey and Mönig 2010].

### 3.6 Concurrency

Concurrency (or "multi-threading") is often considered an advanced programming technique. Yet our everyday world is highly concurrent, so Scratch users are not surprised that a sprite can do several things at once. For example, in a simple paddle game, one script might move the ball, another might make it bounce when it hits the paddle, and another might end the game when the player misses the ball (Figure 9). In Scratch, all of these scripts can run concurrently.

Scratch lacks the explicit concurrency control mechanisms often found in other programming languages, such as semaphores, locks, or monitors. Instead, Scratch builds concurrency control into its threading model in a way


Fig. 10. A simple synchronization lock in Scratch.

that avoids most race conditions, so that users do not need to think about these issues. This is done by constraining where thread switches can occur.

In many preemptive threading models, a thread switch can occur between any two instructions. In the Scratch model, a thread switch can occur in only two places: (1) on a command that waits explicitly (e.g., "wait 1 second") or (2) at the end of a loop. A thread switch cannot occur in the middle of a sequence of non-waiting statements, or between the test of an "if" command and its body.

The Scratch threading model allows users to reason about a script in isolation, giving little or no consideration to potential side effects from the interleaved execution of other scripts. For example, in the code shown in Figure 10, the command that sets the "busy" variable is guaranteed to be executed immediately after the test, without any other thread getting control. As this example shows, the Scratch threading model supports a kind of implicit "critical section" that, in theory, allows the user to create their own synchronization mechanisms. In practice, such mechanisms are seldom necessary.

Although the Scratch threading model avoids most race conditions, it does not eliminate all concurrency issues. The most common one that arises in Scratch is when multiple scripts are triggered by an event or broadcast, and the ordering of those scripts is not what the user expects. However, once users understand that multiple scripts triggered at the same time are run in an arbitrary order, they readily understand the solution, which is to have the event trigger a single script which then triggers the other scripts in the desired order.

## 4. COMPARISON WITH ALICE AND GREENFOOT

Like Scratch, Alice, and Greenfoot are intended to introduce programming to those without prior experience and, as a result, the three systems share many of the same design goals. For example, all three systems support rich graphics and sound and let users create projects that connect to their interests.

Both Alice and Greenfoot target older students than Scratch, introduce class-based object-oriented programming, and emphasize Java or Java concepts, making them suitable as preparation for the Advanced Placement Exam in Computer Science. Greenfoot, since it is Java, also allows students to explore high-performance computation (e.g., complex simulations or cryptography problems) and to extend the system. Alice is the only one of these systems that supports 3-D graphics.

Scratch targets younger users than the other two systems, focuses on self-directed learning, and emphasizes tinkerability. While all three systems allow media to be imported, Scratch includes tools to draw images and record sounds.

The 2-D images used by Scratch and Greenfoot are easier to create and edit than the the 3-D models used by Alice. The Scratch command set is designed to encourage a wider variety of project types than the other two systems, and the ability to easily share projects on the Scratch Web site, with its active user community, provides motivation and opportunities to learn from others.

Scratch has been used as an introduction to both Alice and Greenfoot. The transition from Scratch to Greenfoot is especially smooth, since both are 2-D systems and Scratch's sprites and stage map directly to Greenfoot's actors and world.

## 5. CONCLUSIONS

The Scratch programming system strives to help users build intuitions about computer programming as they create projects that engage their interests. The user interface layout, with its prominent command palette and central scripting area, invites users to program. The Scratch blocks language eliminates syntax errors, allowing users to focus on interesting problems right away, rather than struggling merely to get their program to compile. Block shapes and visual feedback while dragging help the user learn how to assemble programs and use data types. Runtime error messages are avoided through failsoft commands, while a carefully designed concurrency model avoids race conditions.

The Scratch programming language emphasizes simplicity. The type system and object model were designed to work smoothly without prior explanation, yet to make perfect sense on closer examination. Scratch has a surprisingly small number of commands, especially given the wide range of project types it supports. A loosely-coupled inter-object communication system allows sprites to be exchanged without breaking dependencies, fostering collaboration and code sharing.

The system is always live, with no run/edit switch, so commands or code snippets can be run with a click, and graphical feedback shows execution. Variables and lists have concrete visualizations, so the effect of data operations can be seen immediately. These features support and reward discovery through tinkering.

The Scratch programming environment and language work together to create a system that is exceptionally quick to learn—users can be programming within fifteen minutes—yet with enough depth and variety to keep users engaged for years.

## REFERENCES

diSessa A. and Abelson, H. 1986. Boxer: A reconstructible computational medium. *Comm. ACM 29*, 9, 859–868.

Harvey, B. and Mönig, J. 2010. http://byob.berkeley.edu/ (accessed 6/10).

Kafai, Y. and Resnick, M., eds. 1996. *Constructionism in Practice: Designing, Thinking, and Learning in a Digital World*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ.

Kay, A. 2010. Squeak etoys, children, and learning. http://www.squeakland.org/resources/articles/ (accessed 6/10).

Malan, D. and Leitner, H. 2007. Scratch for budding computer scientists. *SIGCSE Bull. 39*, 1, 223–227.

MALONEY, J. AND SMITH, R. 1995. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST'95)*. 21–28.

PAPERT, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. BasicBooks, New York.

RESNICK, M., KAFAI, Y., MAEDA, J., ET AL. 2003. A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically-disadvantaged communities. Proposal to the National Science Foundation (project funded 2003–2007).

RESNICK, M., MALONEY, J., MONROY-HERNÁNDEZ, A., RUSK, N., EASTMOND, E., BRENNAN, K., MILLNER, A., ROSENBAUM, E., SILVER, J., SILVERMAN, B., AND KAFAI, Y. 2009. Scratch: Programming for all. *Comm. ACM 52*, 11, 60–67.

STEINMETZ, J. 2002. Computers and squeak as environments for learning. In M. Guzdial and K. Rose, Eds., *Squeak: Open Personal Computing and Multimedia*, Prentice-Hall, Inc., Upper Saddle River, NJ. 453–482.

WEGNER, P. 1987. Dimensions of object-based language design. *SIGPLAN Not. 22*, 12, 168–182.

Received August 2010; accepted September 2010